Fault-Tolerant Systems Design – Estimating Cache Contents and Usage[1]

Authors: Raphael R. Some Raphael.R.Some@jpl.nasa.gov, John Beahan John.Beahan@jpl.nasa.gov, Garen Khanoyan Garen.Khanoyan@jpl.nasa.gov, Leslie N. Callum Leslie.N.Callum@jpl.nasa.gov
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove
Pasadena, CA 91109
(818) 354-1902

Abstract—Development of the REE Commercial Off The Shelf (COTS) based space-borne supercomputer requires a detailed knowledge of system behavior in the presence of Single Even Upset (SEU) induced faults. When combined with a hardware radiation fault model and mission environment data in a medium grained system model, experimentally obtained fault behavior data can be used to: predict system reliability, availability and performance; determine optimal fault detection methods and boundaries; and define high ROI fault tolerance strategies. The REE project has developed a fault injection suite of tools and a methodology for experimentally determining system behavior statistics in the presence of SEU induced transient faults in application level codes. Where faults cannot be directly injected, analytic means are used in conjunction with experimental data to determine probabilistic system fault response. In many processors, it is not possible to inject faults directly into onboard cache. In this case, a cache contents estimation tool can be used to define probabilistic fault susceptibility which is then combined with direct memory fault injection data to determined fault behavior statistics. In this paper we discuss the structure, function and usage of a PPC-750 cache contents estimator for the REE project.

TABLE OF CONTENTS

1. Introduction

The objective of the Remote Exploration and Experimentation (REE) Project is to bring supercomputing technology into space. It has twin goals of i) demonstrating a process for rapidly transferring Commercial-Off-The-Shelf (COTS) high-performance computing technology into ultra-low power, fault tolerant architectures for space and ii) demonstrating that high-performance onboard processing enables a new class of science investigation and highly autonomous remote operation.

The REE project is employing mainly COTS hardware and software components, and relying on Software-Implemented Fault Tolerance (SIFT) to mitigate the effects of radiation-induced errors. Natural space radiation can cause soft errors known as SEU's in non-radiation-hardened electronics. Thus, REE's primary reliability concern is the detection and mitigation of SEU's. To provide ease of mission insertion, flexibility in configuration, straightforward upgrade as the state of the art progresses and ease of fault tolerance insertion, REE's architecture of choice is a cluster computer. The intent is to leverage the considerable technology investments made in commercial cluster computers and to augment or modify the standard commercial cluster architecture as necessary to provide enhanced reliability for the embedded spaceborne environment. A cluster computer is a parallel processing system which consists of interconnected stand-alone computers working together as a single integrated computing resource [1]. Some of the salient characteristics of cluster computers are: multiple high performance processors with local memory, fast network interconnects, high-bandwidth/low-latency communication protocols, a standard Operating System (OS) on each processor, access to shared mass storage and a convenient parallel programming environment. Figure 1 shows the baseline REE architecture comprising a set of processing and mass memory nodes which are multiply interconnected via a high speed switched network fabric.

Each processing node comprises two state of the art processors, eg. PPC750 or G4 processors, a large local memory and a network interface. In addition to network low level interconnect functions, the network interfaces also provide high level I/O handling protocols such as Message Passing Interface (MPI) so that the node processors may be off-loaded from relatively mundane I/O tasks. The mass memory nodes provide a large (several GigaBytes), solid state, reliable, non-volatile memory for the system.

Multiple such "disk emulators" are provided for parallel high-speed access by the processing nodes as well as for fault tolerance support. The spacecraft interfaces to the REE computer via these mass memory nodes through the inclusion of a custom I/O controller in each mass memory node which provides multiply redundant interconnection to the spacecraft data and housekeeping busses. Thus, the spacecraft control computer and instrument controllers view the REE cluster as a mass memory device. Instrument data and processing commands are written, as files, to the mass memory, while processed data and status are accessed as file read operations. The spacecraft housekeeping bus is extended into the individual nodes of the REE computer to facilitate externally commanded diagnostic procedures by the Spacecraft Control Computer (SCC). Advanced processor architectures are increasingly implementing low level fault detection/protection mechanisms and some of these mechanisms are baseline into the REE architecture. These include Single Error Correct Double Error Detect (SECDED) Error Detection and Correction (EDAC) on local memories, parity protection on external caches, exception detection in ALUs and MMUs, and watchdog-configurable timers.

In addition to the dual processors, each node has 1MB of L2 cache for each processor and 256MB of shared memory. The Mass Memory Nodes are as described above. The internal REE system interconnect is a dual redundant Myrinet switched network fabric with a maximum theoretical bandwidth of approx. 1.23 Gb/S bidirectional per channel. Network topology is a chordal ring an (octal) network switch located in each node. The Mass Memory nodes provide dual redundant 1394 interfaces for normal operation and a dual redundant IIC "backdoor" bus for diagnostic, debug and housekeeping functions.

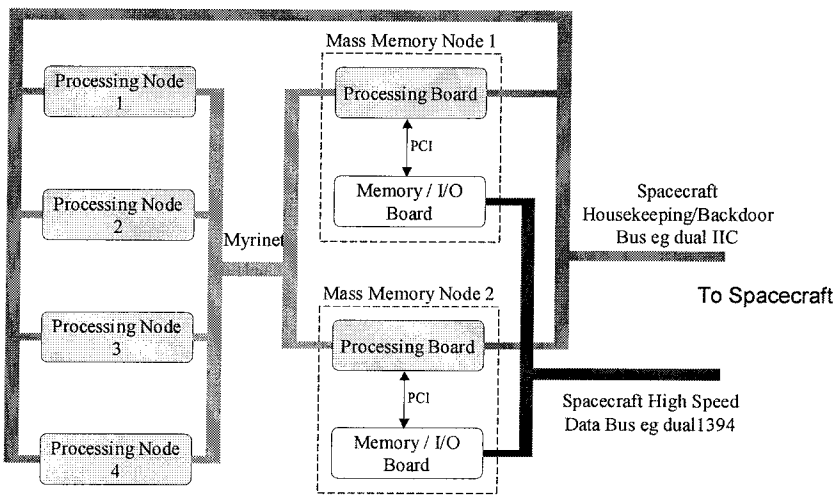


Figure 1: REE Baseline architecture useable if integrated with spacecraft

The development of an effective and efficient fault detection and mitigation strategy for REE requires a detailed knowledge of fault types and rates and of fault propagation paths, behaviors and probabilities. These characteristics are dependent on the hardware used to implement the architecture, the radiation environment, the system and application software, and their interactions. In order to investigate these issues, a method has been developed that utilizes experimentation to obtain fundamental effects and modeling to predict system level behavior, performance, reliability and availability. The following section explains the methodology and tool set.

2.Methodology Overview

The REE project requires a means for trading off performance and power utilization versus reliability and availability. The method must be generally applicable to alternative architectures and applications and, once developed, relatively straightforward to implement. Unlike traditional fault tolerant systems, a degree of unreliability or unavailability is acceptable in many applications, i.e., .95 or .99 rather than .99999 may be an acceptable reliability figure for some REE missions. On the other hand, it is imperative that the system fault behavior and reliability be accurately predictable. The mission system engineer must be able to 'dial in' a desired level of reliability and fault behavior based on mission phase and criticality. Thus, a methodology is required which will allow characterization and modeling of probabilistic system behavior, reliability and availability under varying applications, environments, loads, and operational scenarios.

Radiation effects experiments are performed on the hardware components to determine subsystem level radiation sensitivities. Results for a processor, for example, include fault rates for the L1 data cache, the L1 instruction cache, the general-purpose registers (GPRs), the floating

point registers (FPRs), the memory management unit (MMU), etc.

The results of the radiation experiments are used to develop a radiation fault model [2]. This model is used to predict the fault rates that will occur in a given radiation environment (e.g., Low Earth Orbit, Geo-synchronous Orbit, Deep space, Solar Flare, etc.). The model provides the number of faults per unit time per subsystem.

Using information about the hardware architecture, the Error Model predicts the types of errors that can arise as a result of an SEU occurring in a given subsystem. Essentially, the process of generating the error model is one of listing all possible faults and then, by analysis, propagating each fault through the hardware to the first point at which it impacts software or system operation. The emphasis of this effort is on subsystems into which faults cannot be directly injected with Software Implemented Fault Injection (SWIFI). Thus, it is not necessary to trace every possible error resulting from a general -purpose register bit flip. It is however, necessary to list all the possible outcomes of SEUs in MMU and cache address translation registers, cache tag rams, etc.

The Hardware Utilization Model, provides a means for determining the software (hardware utilization) dependent probabilistic fault propagation statistics and the method by which SWIFI fault injection techniques [3], [4] can be used to emulate the effects of the underlying fault.

The central component of this methodology is the construction and execution of fault injection campaigns. Fault injection campaigns are designed to provide fault/error effects of the faults (e.g., system crash/hang, incorrect result, no apparent effect) and their associated probabilities. The Cache Contents Estimator (CCE), which is the focus of this paper, is used to deal with the inability of SWIFI techniques to inject bit flip faults into the processor's cache memories. Faults are injected into an application's instruction, data, heap, and stack segments in main memory to determine the fault behavior statistics of each type of error. The CCE predicts how much of each of these segments will be in the cache at any given time. The final error rate for each of these segments in cache is proportional to its size. The system model combines the behavior statistics from memory fault injection experiments with the cache fault rate calculated by the fault model to determine overall system failure rate due to cache SEUs.

Finally, the system reliability and performance model is constructed using knowledge of the system architecture, predictions from the fault model, the results of the fault injection experiments and the CCE results. The model predicts the system's reliability and performance in a given radiation environment. It can be used during system development to identify appropriate system architectures and fault tolerance strategies. During fielded operation, the model can be used to predict the system's behavior in changing circumstances and modify it as appropriate (e.g., increase checkpointing frequency, change system operational mode between simplex and various levels of redundancy, uplink fault-tolerant linear algebra libraries, etc.). Once the basic system model has been created and validated, it is relatively straightforward to apply it to a predicted environment. By inputting application-software-specific fault behavior statistics and mission environmental parameters the model will provide the predicted system fault

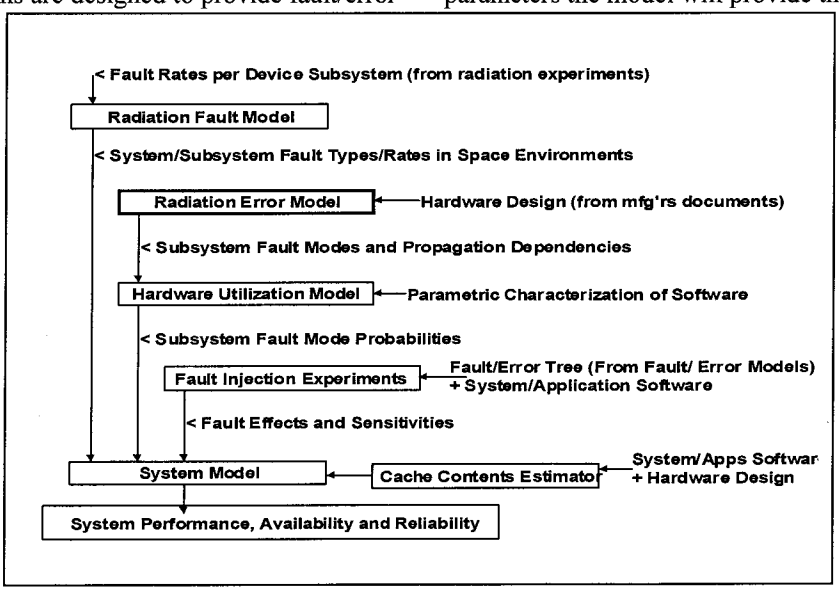


Figure 2: Modeling Methodology Block Diagram

sensitivities of the system components and system fault behavior statistics. The campaigns are conducted on the operational system. Results are analyzed to determine the behavior and reliability for a range of fault tolerance techniques, consequently providing an early testbed for supercomputer based mission development. Figure 2

captures the methodology and tool set developed for the REE project.

3. CCE Goals and Raison D'être

State of the art processor architectures implement one or more levels of on-chip cache memory. These devices do not support software implemented fault injection into their cache memories. The internal cache memories are often, however, the most SEU-vulnerable computer subsystem.. To determine the reliability and availability of a PPC-750 based system it is therefore necessary to determine both the effects of an SEU and the probability of an SEU fault occurring. Inasmuch as the effect of an SEU is dependent on its location, it is necessary to perform memory based fault injection campaigns to obtain the fault effect statistics for each software segment. For the REE project, SWIFI techniques were used to perform massive fault injection campaigns targeting each module of the application and, within each module, the code, data, stack and heap segments. For each type of fault injected, i.e., single or multiple bit flips into code, data, stack or heap, the resultant output was classified as correct, incorrect, crash or hang. Combining the results of these experiments, a statistical distribution was built relating fault type and location to probabilistic result. These fault behavior statistics, however, are not an indication of in-situ system behavior as the main memory is protected from single and double bit errors and the L2 Cache is protected from single bit errors. In a fielded PPC-750 based system, only the internal L1 cache is vulnerable to SEU faults. To obtain realistic estimates of in-situ system behavior, we can combine the fault effect data from memory fault injection experiments with fault arrival rate for the L1 cache. To do this correctly, however, it is necessary to map the contents of the L1 Cache over time and to weight the behavior statistics accordingly in the system model. The CCE provides the cache contents mapping over the execution interval for the system model.

4. Cache Content Estimator Overview

The CCE toolset is divided into two parts: a) Dynamic Application Address Extractor (DAAX) and b) CacheSim. DAAX extracts the memory locations accessed by an application and feeds that data to CacheSim which simulates the content of the L1 cache and provides cache statistics of the executing application.

4.1 DAAX

DAAX is the front-end tool that produces the input for the CacheSim. It captures the instruction stream by stepping through the program using the GDB debugger. For each instruction the tool then reports the Program Counter (PC) value and, for load/store instructions, determines the memory reference virtual address. DAAX is written in Expect. A user supplied configuration file defines the range of code to be captured and an output file specification.

Figure 3 illustrates a block diagram of DAAX user interface.

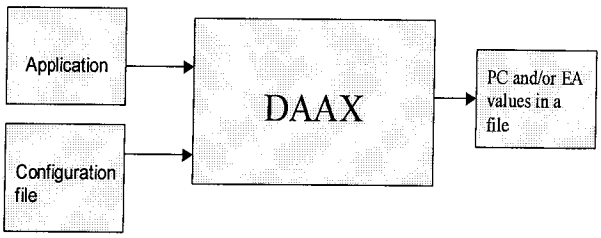


Figure 3: DAAX interface block diagram

4.2 CacheSim

CacheSim provides insight into the contents of a processor's L1 cache by mimicking what the cache controller would do as an application is executed. Currently the tool has the capability of simulating the caches of several members of the PowerPC family.

The PowerPC's L1 cache is divided into two parts: instruction cache and data cache. References accessed via the program counter are processed by the instruction cache and references accessed via the currently executing instruction are processed by the data cache.

Input to CacheSim is provided by the DAAX and the user as shown in Figure 4. This input is the value of the PC address

and the effective address of any accessed data. CacheSim can simulate a virtual memory range of size 2^{32} bytes,

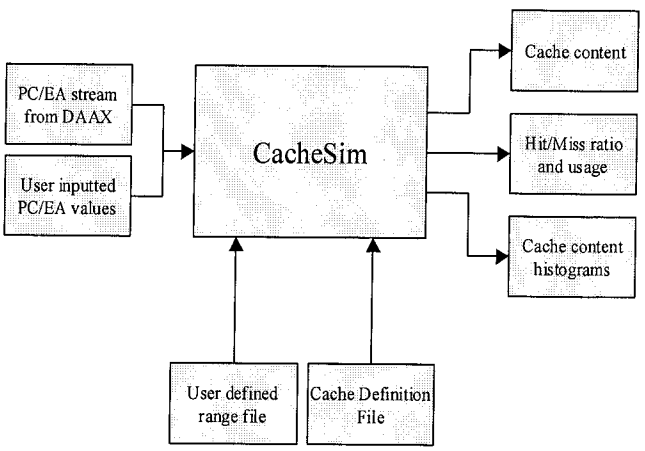


Figure 4: CacheSim interface block diagram

allowing the user to simulate virtual memory addresses as well as physical addresses if desired.

The current toolset only deals with virtual memory addresses due to considerations of practicality and utility. Simulating cache in the virtual memory domain is straightforward, and more easily understood by most programmers and test engineers. Virtual addressing is also used by the JIFI (JPL's Implementation of a Fault Injector)

tool set as well as the system model, thus maintaining a consistent system view across the tool set.

On all current PowerPC processors, the instruction and data caches are nearly identical in configuration and functionality, hence the current design of the tool similarly implements a symmetrical cache architecture. The difference between the data and instruction caches is reflected in the valid state bits. The CacheSim ignores these bits and implements a write thru mode cache. The initial release of the tool was geared towards the immediate needs of the REE project and focused on the PPC-750, which is the default configuration, but considerations were taken to design the tool in such a manner as to support future upgrades. CacheSim can simulate different types of processor caches by means of a custom cache configuration file. This configuration file provides the tool with all the parameters required for modeling the L1 cache such as cache size, number of sets and degree of associativity.

During the simulation, CacheSim provides access to the full contents of both caches, the ongoing hit/miss ratios, the usage of each cache (percentage of cache that contains valid or active data), and histograms of the cache content. By providing a memory range file, the user can get customized histograms that reflect which ranges of memory are active in the cache.

5. Detailed Design and Operation

5.1 DAAX

DAAX runs the specified application on a single node under GDB. It gathers PC and memory reference information while stepping through the application. The parameters for DAAX are specified in a user supplied configuration file. The configuration file has as parameters the start and stop points of the extraction, the application and any required application input parameters. The application will run in normal mode until the start point is reached. At this point DAAX begins stepping through the application. A loop is entered in which, for each instruction, the address information is extracted until either the count parameter (also set in the configuration file) or the stop point is reached.

Within the loop, the following actions are performed: GDB prints the current instruction in GDB's standard format. The instruction is then decoded to see if it is a load/store or another type of instruction. If the instruction is not a load or store, then the PC value is written to the output file. If the instruction is a load or as store, then the effective address of the memory reference is calculated and written to the output file along with the PC value. There are two options for the addressing mode: indexing and summation. If the instruction is a store or a load (i.e., a read or write from or to a memory location), then the effective address is calculated by reading the registers referenced in the instruction and performing the indicated operation, i.e., summation or index. The final step in the loop is to compare the current PC value to the stop value in the user configuration file. If these values are equivalent then the loop is ended and the application's normal running sequence is resumed until the application ends.

The final result is the DAAX output file consisting of an ordered listing of PC and (where applicable) virtual memory address references, with all loops unrolled. The flow of operation for the DAAX tool is illustrated in figure 5.
The DAAX input parameters are illustrated in table 1.

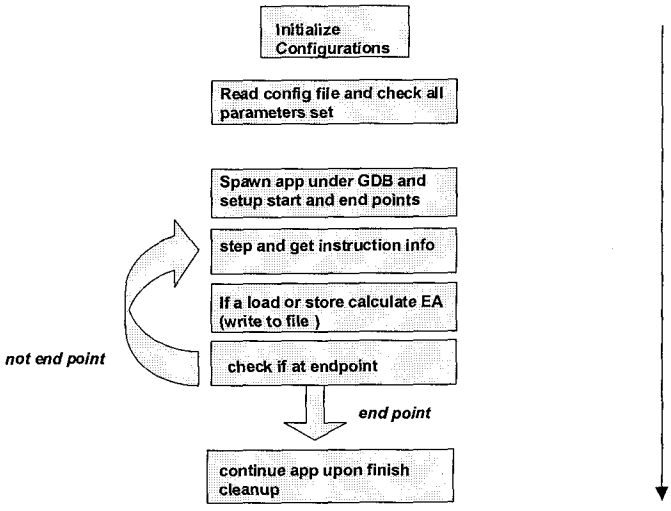


Table 1 DAAX parameters

Parameters required by the tool	
count	number of instructions to step through
stepval	the step unit
break_start	location where stepping starts
break_end	location where stepping stops
logfile	on/off capturing output into a log file
outputfile	on/off the printing of EA and PC in a file
cache_sim	on/off output file for the cache simulator
appparams	input arguments for the application
appname	name of the application executable
apppath	the path to the application executable

The output file will be of the format shown in table 2.

Table 2: Sample DAAX output

Example Sample Output format	
PC	EA
0x100066dc	0x7ffff66c
0x100066e0	
0x100066e4	0x7ffff674
0x100066e8	0x7ffff674
0x100066ec	0x7ffff614
0x100066f0	

5.2 CacheSim

Figure 6 is a functional block diagram of CacheSim. The heart of the tool consists of the Simulation Core, which updates the Virtual Cache based on the PC and Effective Address (EA) inputs. These inputs (PC and EA), as explained earlier, are obtained from the DAAX or entered manually via the user interface. The output of the CacheSim is a statistical analysis of the cache contents and operation including hit/miss statistics and cache contents by user definable region in histogram form. The User Interface provides access and control of the output functions as well as specification of the Simulation Core operational parameters. The User Interface also allows the user to print out the contents of the Virtual Cache.

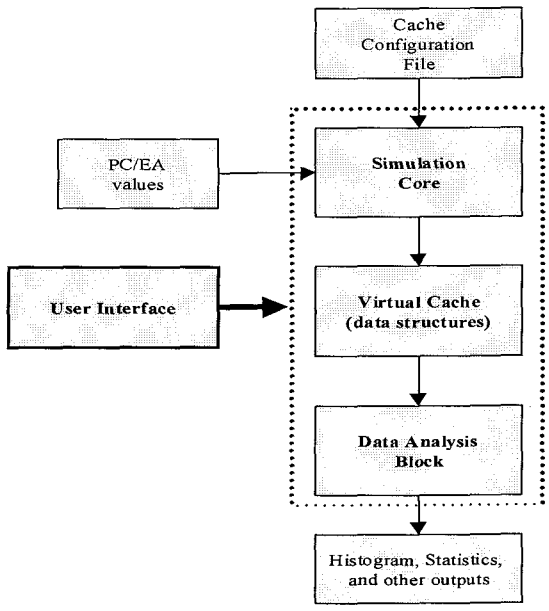


Figure 6: Functional block of the cacheSim

The Cache Configuration File is an input to the Simulation Core, which determines how the cache is structured. The cache structure, in turn, determines how the input values (PC and EA) will be parsed into tag address, set number and block size. The parameters also determine the depth of the associativity field. Figure 7 shows the PPC-750 cache organization. Table3 shows the cache parameters for the PPC-750.

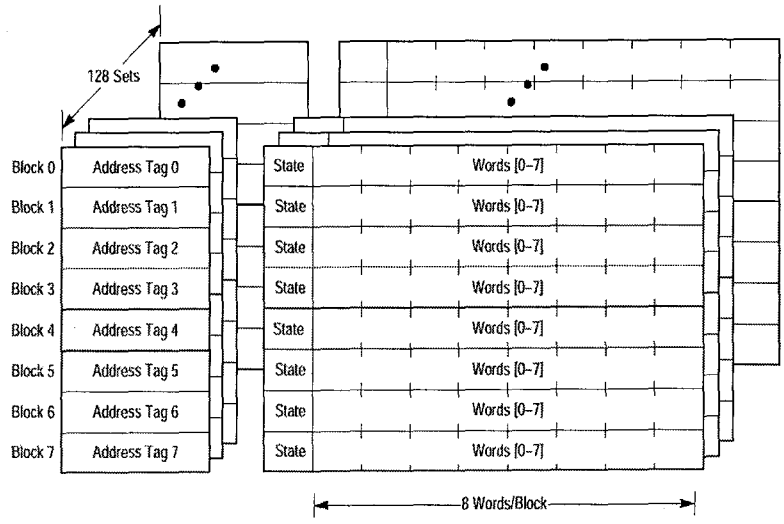


Figure 7: Cache configuration of PPC-750

Table 3: Cache Configuration File Parameters

Parameter	Value
Sets	128
Associativity	8
Block size (words)	8
Word size (bytes)	4

The cache size (in bytes) is determined by the product of the cache parameters in the Cache Configuration File.

$$cache\_size = (S * A * B * W) \quad (1)$$

where S is the number of sets, A is the set associativity of the cache, B is the cache block size in words and W is the word size in bytes. The case shown above, for example, will have a cache size of 32 Kbytes (128*8*8*4).

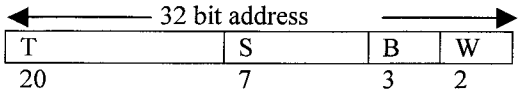
The length of the tag field "T" (in bits) is also set by the cache parameters.

$$tag\_field\_length\ T = 32 - \log_2(S) - \log_2(B) - \log_2(W) \quad (2)$$

The tag size in this case is 20 bits (32-7-3-2).

Table 4 shows the memory address field parsed into its T, S, B and W components.

Table 4: Memory Address Field



The Virtual Cache data structure is created, by the Simulation Core using the cache parameters from the Cache Configuration File as explained above.

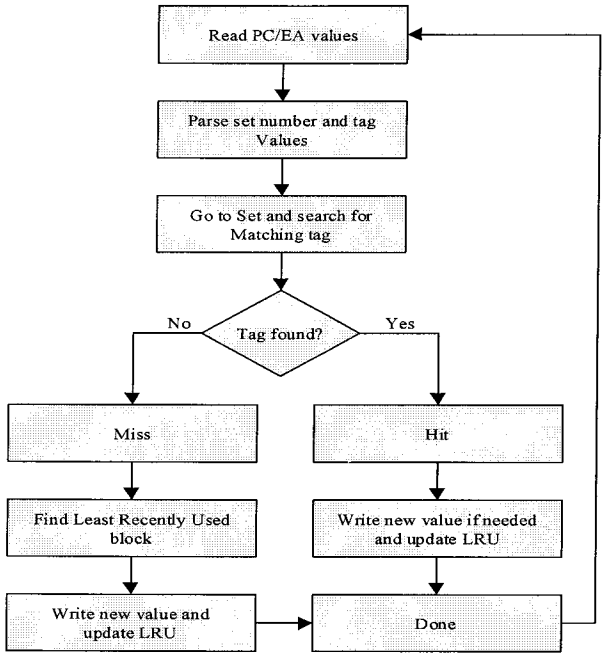


Figure 8: Operational diagram of the Simulation Core

The simulation core is the main component of the CacheSim. Figure 8 illustrates the behavior of the simulation core. It processes each PC and EA value independently. The EA is optional as there are many instructions that solely operate on registers, thus if an EA value is not provided, the data cache will not be updated. The PC and EA are parsed into set number, tag value and block size as illustrated above in table 4. The address tag is searched for in the set specified by the set number. If the tag exists then address reference is considered a hit, and the Virtual Cache contents and hit/miss statistics are updated. If no tag is found the address reference is considered a cache miss and the Virtual Cache contents and statistics are updated. For the case of a cache miss the least recently used block is determined using the Least Recently Used (LRU) algorithm.

The LRU method is common with all the PowerPC processors. A few PowerPC processors use a modified algorithm known as pseudo-LRU. This modified version is meant to speed up the process of deciding which block to replace, however, with respect to its functionality, the optimized method is very close to the standard LRU algorithm. For our purposes, this estimation is sufficiently accurate. The LRU algorithm is implemented by sequentially numbering each memory access. Once a set is full, the block replaced is the one with the lowest number. If an existing block is being accessed again, i.e., in the case of a cache hit, its old access number is replaced with the access number of the new memory reference. In the case of a cache miss the least recently used block (access number, tag field...) is found and updated along with the hit/miss statistics.

The process of updating the cache is the same whether the processor is requested to read or write from that location.
The tool currently assumes the processor's cache is in write-through mode, i.e., every update to the cache is immediately written to the main memory. This eliminated dealing with cache coherency operations and simplified the model and was sufficiently accurate for our needs. The Data Analysis Block performs statistical analysis of the cache usage data as the simulation progresses. The tool tracks both cache hits and misses. These values along with current count of valid cache items are available through out the simulation. A histogram function is also available to the user to track the active contents of the cache. The histogram feature calculates the number of active cache blocks in a specified address range. The user can specify single or multiple ranges, save a formatted file of the entire content for later analysis, or quickly view the preset range statistics. The contents of both caches can be saved or printed to the screen for the users evaluation. The entire contents of the Virtual Cache can also be saved to a binary file that can be restored in later sessions.
An automation feature provides the ability to take a snapshot of the cache statistics at intervals for use in creating a cache time line. The cache timeline can be used as an input to the system model or as a tool to assist in application or system profiling.

An example of the CacheSim input and output files is shown in Figure 9. The CacheSim input file is the DAAX output file. The CacheSim output file shows a snapshot of the statistics in time as the input file is processed. In this sample, a portion of an application was processed by DAAX and the results were fed into CacheSim. The CacheSim output is the statistical data on the content of both caches after 500 instructions were simulated on a "fresh" cache. Out of the 500 instructions, 215 requested memory accesses. The corresponding hit/miss ratios are displayed. The active content distribution of each cache is also displayed grouped by user defined memory ranges.

Sample DAAX Output

```
0x10009044 0x20003504
0x10009048 0x200034e4
0x1000904c
0x10009050
0x10009054
0x10009058
0x100090c8 0x200034e0
0x10009058
0x100090c8 0x200034e0
0x100090cc
0x100090d0
0x100090d4
0x100090d8
0x100090da
0x100090dc
...
0x1000be48
0x1000be4c
0x1000b9f0 0x7fffed64
0x1000b9f4 0x7fffed70
0x1000b9f8 0x7fffed00
...
```

Sample CacheSim Output

```
Instruction Cache
=========================================================
           main: (0x10001000 - 0x10007f00) :   42 ( 4.10%)
            fft: (0x10012320 - 0x10018900) :    0 ( 0.00%)
          gabor: (0x10008900 - 0x10009000) :    0 ( 0.00%)
          Empty: (                       ) :  975 (95.21%)

Data Cache
=========================================================
           heap: (0x20002000 - 0x20050000) :    0 ( 0.00%)
           data: (0x20000000 - 0x20002000) :   17 ( 1.66%)
          stack: (0x7fff0000 - 0x7fffffff) :   36 ( 3.52%)
          user1: (0x20000000 - 0x40000000) :   17 ( 1.66%)
          Empty: (                       ) :  969 (94.63%)

             count          hit(%)     miss(%)
=============================================
Inst: |            500  |   90.20  |    9.80
Data: |            215  |   89.00  |   11.00

Inst cache usage:   4.79% full
Data cache usage:   2.37% full
```

Figure 9: Sample from an actual run

6. Conclusions and future work

The CCE was built to provide insight into the cache contents over time during execution of applications on the REE cluster computer. It provides a reasonable level of simulation fidelity for the PPC-750 L1 Cache. The tool is modular and flexible and can be easily ported to other processors and processing systems.

There are certain limitations to the current tool including the fact that it ignores the "valid" bits and implements only a write through mode. This is of minimal impact to the current REE project but maybe significant at a later time. With this exception, the tool is relatively general purpose and we anticipate that it will be used on future computer development projects. Future work may include handling other cache modes and implementing the valid bits and associated algorithms.

CacheSim's operation and results were verified by manually stepping instructions and verifying that the correct block in the correct set was being updated and replaced if necessary. The resulting statistics were also manually verified to ensure correctness. A more thorough method of verification would be to compare the results of a sample set of instructions executed on a PowerPC and simulated on CacheSim. However gaining insight to the content of L1 cache inside the processor is a complicated task. This type of verification remains as future work to be done.

Currently, there is no API or automated interface between the DAAX and the CacheSim. A script will be written in the future to provide this user-friendly feature. Similarly, there exists no defined interface between the CCE and the System Reliability Model. Future work may include such an interface.

The most pressing issue for future work is the extension of the CCE to capture OS level execution. The CCE currently looks only at application level code. Future work will include extending its capabilities to kernel or OS level code.

REFERENCES

[1] R. R. Some and D. C. Ngo, "REE: A COTS-Based Fault Tolerant Parallel Processing Supercomputer for Spacecraft Onboard Scientific Data Analysis," Proc. of the Digital Avionics System Conference, vol. 2, pp. B3-1-7 - B3-1-12, 1999.

[2] J. J. Beahan, L. Edmonds, R. D. Ferraro, A. Johnston, D. Katz, R. R. Some, "Detailed Radiation Fault Modeling of the Remote Exploration and Experimentation (REE) First Generation testbed Architecture," Aerospace Conf. Proc., vol. 5, pp. 279-291, 2000.

[3] R. Some, A. Agrawal, W. Kim, L. Callum, G. Khanoyan, A. Shamilian, A. Nikora, " Fault Injection Experiment Results in Space borne Parallel Application Programs," IEEE Aerospace Conference, 2002

[4] R. Some, W. Kim, G. Khanoyan, L. Callum, A. Agrawal, J. Beahan "A Software-Implemented Fault Injection Methodology for Design and Validation of System Fault Tolerance," Int. Conf. On Dependable Systems and Networks (DSN'2001), Göteborg, Sweden, July 2001

[5] J. J. Beahan, "SWIFI: A Software-Implemented Fault Injection Tool," JPL Internal Document, June 2000.

Raphael Some

John Beahan

Garen Khanoyan *is a systems test engineer on the REE project at Jet Propulsion Laboratory. He has developed, along with other system test team members, a method and a tool set for conducting fault injection campaigns. He has a BSEE from University of Southern California.*

Leslie Callum *is a system test engineer on the REE project at Jet Propulsion Laboratory. She has developed, along with other system test team members, a method and a tool set for conducting fault injection campaigns. She has a BSEE from University of California Los Angeles.*

Acknowledgment
This work was performed at the Jet Propulsion Laboratory, California Institute of Technology under a contract with the National Aeronautics and Space Administration. This project is part of NASA's High Performance Computing and Communications Program, and is funded through the NASA Office of Space Sciences.